UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

Análisis del uso de metodologías de Integración y Despliegue Continuo en aplicaciones web

Autor/es

ÓSCAR MARTÍNEZ MARTÍNEZ

Director/es

JESÚS MARÍA ARANSAY AZOFRA

Facultad

Escuela de Máster y Doctorado de la Universidad de La Rioja

Titulación

Máster Universitario en Tecnologías Informáticas

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2018-19

publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

Trabajo de Fin de Máster

Análisis del uso de metodologías de Integración y Despliegue Continuo en aplicaciones web

ÓSCAR MARTÍNEZ MARTÍNEZ

Tutor: JESÚS M. ARANSAY AZOFRA

MÁSTER:
Máster en Tecnologías Informáticas (853M)

Escuela de Máster y Doctorado

AÑO ACADÉMICO: 2018/2019

Resumen

El presente Trabajo Fin de Máster tiene como objetivo el estudio de las metodologías y herramientas más actuales de Integración y Despliegue Continuo.

A lo largo del estudio iremos planteando distintos escenarios y posibilidades que se pueden dar cuando trabajamos con estas metodologías. Propondremos diversas alternativas para dar solución a los casos que vayan surgiendo. Expondremos un caso de uso real de una aplicación web que tenemos que desarrollar consistente en un portal web para los empleados de una organización. Iremos recorriendo los distintos escenarios relacionados con la Integración Continua, Entrega Continua y Despliegue Continuo, escogiendo las técnicas y herramientas que mejor se adapten a cada situación. Abordaremos la gestión de versiones del código de la aplicación con Git y su almacenamiento en GitHub. Veremos cómo integrar GitHub con un servicio de Integración y Entrega Continua que será Jenkins. Comentaremos los tipos de tests que se pueden realizar con Node.js e implementaremos una batería de tests de integración y E2E. Analizaremos e implementaremos entornos de ejecución para *testing* y producción con Docker y por último implementaremos una Jenkins Pipeline con la finalidad de integrar todos los procesos que debe incorporar un ecosistema CI/CD.

Palabras clave: Integración Continua, Entrega Continua, Despliegue Continuo, *testing*, Node.js, Git, Jenkins, Docker, balanceadores de carga.

Abstract

The aim of this Master's Thesis is to analyze the most current methodologies and tools related to Continuous Integration and Continuous Deployment.

Alongside of this study we will present different kinds of scenarios and possibilities that can occur when we work on these methodologies. We will purpose a wide variety of alternatives in order to find a solution to distinct cases. We will expose a real use case of a web application that we need to develop. We will go through different stages related with Continous Integration, Continous Delivery and Continuous Deployment, choosing techniques and tools that best suit each situation. The web application we are going to develop during this work consists in a web portal for employees of an organization. We will approach the source code version management with Git and its cloud storage on GitHub. We will also see how to integrate GitHub with a Countinous Integration and Continous Delivery (CI/CD) service as is Jenkins. We will analyze the types of tests we can develop on Node.js and we will program some integration and E2E tests. We will implement Docker as a *testing* and production execution environment and finally we will define a Jenkins Pipeline with the goal of integrate every process that should embed a CI/CD system.

Keywords: Continuous Integration, Continous Delivery, Continous Deployment, *testing*, Node.js, Git, Jenkins, Docker, load balancers.

ÍNDICE GENERAL

CAPÍTULO 1. INTRODUCCIÓN

El mundo del desarrollo de software ha incorporado nuevas técnicas y metodologías en los últimos años. Con el auge de las metodologías ágiles, que abogan por un desarrollo iterativo-incremental, y que, por su naturaleza, se basan en constantes cambios y actualizaciones del producto, han surgido paralelamente métodos y herramientas que allanan el camino a la hora de llevar a cabo estos procesos.

Aunque se llevaba hablando de este tipo de procesos desde principios de los años 90 con el surgimiento de las metodologías ágiles, no fue hasta 2006 cuando Martin Fowler [1] acuñó el término Integración Continua como el conjunto de procesos en que miembros de un proyecto software integran frecuentemente sus aportaciones de manera que, a la vez que se trata de integrar un nuevo cambio, de forma automática se verifica el código y el funcionamiento de la aplicación para finalmente liberar y poner en producción la nueva versión lo más rápido posible.

Como extensión de la Integración Continua, y con el fin de automatizar aún más el proceso de integración, surgen también otros términos como Entrega Continua y Despliegue Continuo.

1.1. Integración Continua

La Integración Continua (*Continuous Integration*) es un modelo informático que consiste en realizar cambios en un producto software de la manera más automática posible.

El mayor pilar de la Integración Continua se basa en intentar automatizar el proceso de integración de cambios. De tal manera que el desarrollador puede incorporar pequeños cambios rápidamente. De tal forma que el código pasa rápidamente a producción en lugar de permanecer un largo tiempo en el repositorio hasta que se publique un alto número de cambios en bloque. Esto trae grandes beneficios, pues el código permanece menos tiempo parado y nos permite detectar posibles fallos de forma prematura.

Este proceso está muy ligado con la gestión del código de una aplicación.

1.1.1. Gestión de código

La gestión del código involucra el conjunto de herramientas y estrategias utilizadas a la hora de almacenar y administrar las distintas versiones del código fuente de nuestra aplicación. Controlando las versiones de nuestro código, vamos a poder registrar y administrar los cambios realizados sobre el código de nuestro proyecto, de tal manera que sea posible recuperar versiones especificas más adelante; esto es un principio fundamental de la Integración Continua.

Existen numerosas herramientas para gestión de código y control de versiones. Entre ellas Git, CVS, Subversion, Mercurial o Bazaar. El uso de una u otra depende de las necesidades del equipo de desarrollo y sobre todo de las estrategias que se quieran emplear para gestionar el código.

Cuando gestionamos las versiones del código de una aplicación, es muy importante contar con una estrategia a seguir a la hora de incluir nueva funcionalidad en la aplicación. A continuación, se detallan algunas de las estrategias más utilizadas.

Lineal. Todos los desarrolladores suben las tareas que van acabando en la misma línea o rama. Esta estrategia tiene un gran problema cuando se utiliza para trabajar en equipo porque puede darse el caso de que un programador haya introducido un error en el código y el resto de los programadores utilicen su versión para introducir nuevos cambios. De esta forma se arrastrará el error sucesivamente. Otro de los inconvenientes que tiene esta forma de trabajo es a la hora de mezclar código que fue desarrollado paralelamente, porque puede ser que varios programadores hayan modificado inconscientemente el mismo fragmento de código; en este caso, uno de los cambios será sobrescrito.

Ramificado. Surge para ofrecer una solución a los problemas que ocasionaba la estrategia Lineal en equipos de desarrollo donde colabore más de un programador. Ahora partimos de una versión base de la aplicación. Para cada nueva funcionalidad que vayamos añadiendo, se crea una rama nueva derivada de la rama base y se programa la funcionalidad. Una vez que dicha funcionalidad está lista para ser integrada, se une con la rama principal. Esta estrategia es conocida como *Branch per feature* (rama por característica). Es muy utilizada en equipos compuestos por numerosos programadores.

Existen enfoques que se le pueden dar a estas estrategias cuando trabajamos con Integración Continua. A menudo existe un servidor de Integración Continua, que usualmente está conectado con el repositorio donde graba la herramienta de gestión de versiones. Es muy común que dicho servidor tenga programados una serie de pasos que se ejecutan cada vez que un desarrollador añade una nueva característica.

Integración Continua **y ramificación.** Igual que en la estrategia anteriormente definida, se trabaja por ramas. Cada vez que el desarrollador dispara un cambio en una rama, el servidor de Integración Continua combina temporalmente dicha rama con la rama base o master y construye la nueva versión, acto seguido verifica que se pasan todos los tests. Tras haber pasado los tests con éxito, es frecuente que el siguiente paso sea combinar ambas ramas definitivamente y desplegar la nueva versión pero esto ya depende de cómo se haya descrito el flujo de integración o pipeline; lo veremos más adelante.

1.2. Entrega Continua

La Entrega Continua es el paso que sucede a la Integración Continua. El objetivo principal es el de proporcionar de manera fiable una nueva versión de la

aplicación. Extiende la Integración Continua implementando los nuevos cambios del código en un entorno de ejecución de pruebas. En muchas ocasiones este paso lo dispara manualmente el desarrollador tras haber integrado un nuevo cambio en el proyecto, sin embargo, en ocasiones, cuando utilizamos servidores de Entrega Continua, solemos definir *hooks* o *triggers* que disparan este proceso automáticamente cuando el desarrollador agrega un nuevo cambio.

Si la Entrega Continua se realiza de manera óptima, tras haber puesto en ejecución la nueva versión de la aplicación en un entorno similar al de producción, el equipo de desarrollo podría llegar a disponer de un artefacto listo para su despliegue en producción.

1.2.1. Testing

Las pruebas de software son procedimientos que se ejecutan en una aplicación para verificar la calidad y funcionamiento de un producto software.

En términos generales, realizar pruebas para cualquier proyecto software agrega mucho valor al mismo puesto que facilita su escalabilidad, aumenta la robustez de la aplicación y reduce el tiempo de desarrollo de código funcional.

Si bien es verdad que implica un esfuerzo considerable, los beneficios que se obtienen superan la inversión inicial. En general, la utilidad de los tests se asocia más con la idea de asegurar que se preservan las funcionalidades de un software que esta sujeto a cambios o desarrollo de nuevas versiones y funcionalidades.

Este proceso es de vital importancia si queremos poner en marcha la Entrega Continua, pues nos garantiza (siempre y cuando hayamos desarrollado buenos tests) que el nuevo producto no va a originar ningún fallo al entrar en producción.

Atendiendo a la tipología de los tests, podemos agruparlos en tres grupos:

Test unitario. Este tipo de test se encarga de verificar el correcto funcionamiento de un módulo de código. Normalmente busca una alta granularidad, atomicidad y aislamiento con el fin de evitar el máximo de dependencias posibles entre nuestros tests.

Test de integración. Este tipo de test nos permiten verificar la correcta integración entre varios módulos de la aplicación. Suelen involucrar agrupaciones de tests unitarios.

Test *End-to-End* (E2E). Estos tests simulan el comportamiento de un usuario real. Prueban toda la funcionalidad de la aplicación de principio a fin cubriendo secciones de código que los test de integración y unitarios no son capaces de cubrir. Este tipo de tests involucran tanto *backend* como *frontend*, lo que nos proporciona un mayor control para la validación de integración de distintos componentes. Uno de los inconvenientes al que nos enfrentamos a la hora de ejecutar estos tests en aplicaciones web, es que dependemos de tener un navegador en ejecución y un *driver* que permita a nuestro *framework* de *testing* comunicarse con el navegador.

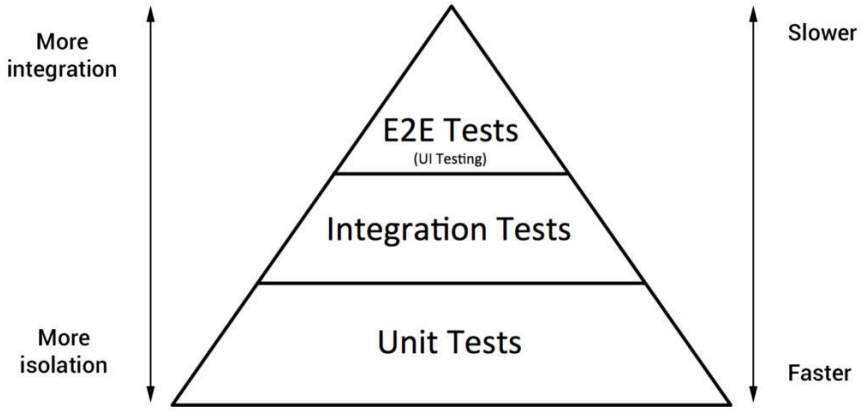


Figura 1. Pirámide de Cohn.

La pirámide de Cohn aconseja [2] que haya mayor volumen de tests unitarios y menor cantidad de tests E2E. La razón es que los tests unitarios son mucho más rápidos de ejecutar y garantizan un mayor aislamiento de funcionalidades. Sin embargo, pese a que los tests E2E son mucho más frágiles y lentos, un correcto desarrollo y mantenimiento de los mismos nos garantizará, con mayor fiabilidad que los test unitarios, que nuestro sistema funciona correctamente.

Cuando trabajamos en Integración Continua, es necesario y casi obligatorio que la fase de *testing* se realice automáticamente, sin previa intervención del equipo de desarrollo. De esta forma la nueva versión de la aplicación se lanzará y posteriormente se ejecutarán los tests sobre ella. Tras conocer los resultados de los tests, se tomará una decisión u otra.

Normalmente el orden de ejecución de los tests suele ser: tests unitarios, test de integración y por último los test E2E.

1.2.2. Gestión de entornos de pruebas

En esta fase entran en juego herramientas que permiten administrar entornos de ejecución, como por ejemplo Puppet, Chef o Ansible. Este tipo de herramientas, también denominadas “de aprovisionamiento”, permiten administrar la configuración de entornos. En general suelen utilizar un lenguaje declarativo para generar un fichero con el cual el usuario puede ser capaz de especificar y describir los recursos de los que debe proveerse el sistema y estados que debe alcanzar. Con esto lo que conseguimos es que todos los entornos que levantemos siempre sean idénticos en varios aspectos: versiones de aplicaciones, configuraciones de red, etc. Por ello, será mucho más fiable desplegar el artefacto en producción tras haber verificado el funcionamiento de la aplicación en un entorno de pruebas similar.

A continuación, se detalla algunos aspectos de las principales herramientas de aprovisionamiento [3].

Puppet. Es código abierto y desarrollado en Ruby. Orientado a administradores de sistemas. Proporciona una alta capacidad de monitorización, muy útil cuando trabajamos con gran número de sistemas. El usuario puede especificar estados finales para las máquinas por medio de ficheros Ruby denominados "*manifest*". Soporta aprovisionamiento cliente-servidor, de manera que los clientes validan, en intervalos de tiempo, su estado actual contra el servidor, el cual dispone de la información del estado que realmente deberían tener.

Chef. Desarrollado en Ruby. Más sencillo que Puppet, pero con menos posibilidades respecto a monitorización. Orientado a desarrolladores software. Permite programar en Ruby los ficheros denominados "*recipes*" que contienen información sobre el estado final que debe alcanzar la máquina.

Ansible. Desarrollado en Python. Orientado a desarrolladores. Al contrario que Chef y Puppet, este no necesita un servidor maestro ni agentes en los nodos. Los ficheros que describen el estado final de la máquina se denominan "*playbooks*" y se escriben en Yaml.

Todas estas herramientas nos proporcionan un modo de orquestar configuraciones de nuestras máquinas, sin embargo, no nos proporcionan realmente el entorno de ejecución como tal. Denominamos entorno de ejecución a cualquier sistema tanto físico como virtualizado que sea capaz de ejecutar un servicio.

Existen numerosas alternativas para generar entornos de ejecución, todas compatibles con las herramientas de aprovisionamiento que hemos visto anteriormente.

Por una parte, tenemos **entornos de ejecución físicos**. Estos entornos de ejecución consisten en poner una máquina física a ejecutar un programa. Esta solución se suele denominar también entorno bare-metal. En definitiva, consiste en arrancar una máquina físicamente e instalar y ejecutar nuestro servicio.

Por otra parte, tenemos **entornos de ejecución virtales**. Estos entornos lo que hacen es abstraer recursos de la máquina física para crear nuevos entornos dentro de dicha máquina. En este caso vamos a distinguir dos tipos de entornos.

Por un lado, tenemos entornos virtualizados de máquinas virtuales como por ejemplo los que nos pueda proporcionar Virtualbox o VMware, y por otro lado tenemos soluciones basadas en contenedores. Los contenedores son una virtualización a nivel de software de un sistema operativo que nos dotan de las dependencias necesarias para que un servicio pueda ser encapsulado y ejecutado dentro del mismo.

La principal diferencia entre máquina virtual y contenedor reside en que los contenedores pueden reaprovechar partes del sistema operativo que hay subyacente, mientras que las máquinas virtuales virtualizan a nivel de hardware y por tanto incluyen al sistema operativo. Tenemos que ver a los contenedores como un conjunto de procesos que se pueden generar y borrar de manera mucho

más ligera que una máquina virtual porque al iniciar un contenedor no necesitamos arrancar un sistema operativo.

Si hablamos de máquinas virtuales e Integración Continua, debemos mencionar a **Vagrant** [4], un gestor de máquinas virtuales que permite creación y configuración de entornos virtualizados orientados al desarrollo de software. Cuenta con un fichero de configuración de máquinas llamado Vagrantfile, en el cual el usuario puede definir la configuración de las máquinas que se van a lanzar.

Cuando hablamos de contenedores es inevitable no dejar de lado **Docker** [5]. Docker es una herramienta de código abierto que permite crear contenedores de aplicaciones. Igual que el resto de las herramientas que hemos explicado anteriormente, Docker también ofrece una manera de aprovisionar máquinas con idénticas configuraciones. Por el contrario, no es tan sofisticada. Docker permite al usuario definir configuraciones y estados de los contenedores mediante unos documentos denominados *Dockerfile*. Estos documentos son interpretados y transformados en imágenes que pueden ser almacenadas y redistribuidas. A la hora de lanzar un contenedor podemos escoger la imagen base, pudiendo lanzar todos los contenedores que queramos utilizando dicha imagen.

1.3. Despliegue Continuo

En Entrega Continua, los nuevos cambios se compilaban y ejecutaban en un entorno de pruebas muy similar al de producción con la finalidad de verificar el correcto funcionamiento de la aplicación. Tras haberse validado todas las pruebas, el desarrollador autoriza (manual o automáticamente) el despliegue en producción de la nueva versión.

Con Despliegue Continuo, el despliegue en producción se realiza de manera automática sin autorización explícita del equipo de desarrollo. Para ello previamente tendremos que haber especificado las reglas necesarias para desencadenar este paso.

Existe una herramienta indispensable cuando hacemos Despliegue Continuo, y es el **balanceador de carga.** Un balanceador de carga es una aplicación que se encarga de enrutar el tráfico entrante a uno o más nodos atendiendo a una serie de reglas especificadas por el usuario. Existen dos tipos principales de balanceadores: de capa 4 y de capa 7. Los balanceadores de capa 4, enrutan el tráfico a niveles de capa de transporte y los balanceadores de capa 7 dirigen el tráfico HTTP/HTTPS. Dependiendo del tipo de reglas que queramos utilizar para balancear las peticiones, utilizaremos un balanceador u otro.

Dependiendo del tipo de aplicación y arquitectura de nodos de la que dispongamos para publicar nuestra aplicación, hemos de escoger la estrategia de despliegue que mejor se adapte. Existen diversas estrategias [6] a seguir cuando tratamos de poner en producción una nueva versión para nuestra aplicación.

Despliegue tipo Big Bang. Como su propio nombre indica, los despliegues se realizan de golpe, actualizando todo el sistema o gran parte del mismo. Esta estrategia de despliegue ha quedado casi en desuso, utilizándose especialmente en ecosistemas poco críticos y proyectos poco complejos. Esta estrategia ha venido fuertemente asociada a lo largo del tiempo a la metodología de desarrollo de software en Cascada, en la cual tenemos flujos de trabajo muy extensos con despliegues esporádicos que liberan grandes cambios en la aplicación.

Despliegue tipo Rolling Upgrade. También conocido como despliegue paso a paso. Es un tipo de despliegue cuyo principio es ofrecer la posibilidad de abortar el proceso de actualización de la nueva versión en el momento que se detecte algún error o incompatibilidad, sin necesidad de provocar una caída de servicio. Este tipo de despliegue es óptimo para arquitecturas en las que contamos con un gran número nodos en los que tenemos replicada nuestra aplicación; en este tipo de topologías, se cuenta con un balanceador que distribuye las peticiones de manera equitativa. Al crear una nueva versión de la aplicación, en el momento del despliegue se va reemplazando gradualmente en cada nodo la versión antigua por la versión nueva; de este modo, mientras un nodo está actualizándose y en consecuencia no puede dar servicio, las peticiones son redirigidas al resto de nodos. Mientras se va ejecutando la actualización gradual, ambas coexisten dando la posibilidad de que los clientes accedan a distintas versiones de la aplicación dependiendo del nodo al que sea balanceada la petición. Si se da el caso de que se detecta alguna incompatibilidad, bastaría con apagar los nodos que alojan la nueva versión de la aplicación y seguir ofreciendo el servicio con la antigua versión hasta que se subsane el fallo.

Despliegue tipo *Blue-Green*. También lo podemos encontrar como despliegue tipo A/B o despliegue tipo *Red-Black*. Como el tipo de despliegue anteriormente comentado, también permite recuperarse de ciertos fallos. En este tipo de despliegue contamos con dos entornos de ejecución idénticos y un balanceador de carga. Uno de ellos (*Blue*) es el nodo que actualmente está alojando la última versión que se liberó de la aplicación y que está soportando todo el tráfico redirigido por el balanceador de carga. En el otro nodo (*Green*) se despliega la nueva versión de la aplicación. Ambos nodos comparten base de datos y configuración de aplicación. Una vez que estamos seguros de que la nueva versión está preparada para dar servicio a los usuarios finales, hacemos que el balanceador de carga pase a redirigir el tráfico al nodo *Green*, de tal manera que la pérdida de servicio es mínima puesto que dura lo que tardamos en reiniciar la configuración del balanceador.

Despliegue tipo *Canary*. Este tipo de despliegue se fundamenta en la liberación de la nueva versión de la aplicación a un subconjunto de usuarios, generalmente poco numeroso, con la finalidad de provocar el menor impacto posible ante cualquier error que pueda traer consigo la nueva versión. Estos usuarios actuarán de “Canario” o *tester* durante un periodo de tiempo. La filosofía es parecida al despliegue *Blue-Green*, sin embargo, aquí van a convivir ambos nodos a la vez. El nodo con la última versión estable (*Blue*) es el que manejará

gran parte del volumen de peticiones y el nodo con la nueva versión (*Green*) es el que recibirá las peticiones de los canarios. Una vez que la versión alojada en *Green* está completamente verificada, el balanceador pasa a redirigir todas las peticiones a la nueva versión.

1.4. DevOps

El término DevOps viene de **Dev**elopment y **Op**eration**s**, que hacen referencia al equipo de desarrollo software y al equipo de operaciones que se ocupan del aprovisionamiento y mantenimiento de sistemas. El principio fundamental del DevOps consiste en crear un ambiente de colaboración entre equipo de desarrollo y sistemas, es decir, que no existan limitaciones de comunicación, tiempos muertos ni bloqueos entre ambos equipos.

Durante años, y actualmente siguen existiendo ambos equipos por separado. Por una parte, tenemos al equipo de desarrollo que se encarga de desarrollar el código y testearlo en sus propias máquinas o entornos que ellos mismos han creado. Y por otra parte tenemos el equipo de sistemas que recibe paquetes de código de los desarrolladores y su misión es desplegarlo en un entorno de producción de la manera más estable posible. A menudo suele ocurrir que el paquete de código pasaba los tests en el entorno de pruebas del desarrollador, pero al ponerlo en producción se producen numerosos fallos; esto hace que el equipo de sistemas malgaste tiempo y se retrase generando un desequilibrio que posteriormente acarreará conflictos y ralentizando la puesta en producción del producto.

En sí, DevOps es un conjunto de buenas prácticas de desarrollo software y especificación de sistemas. La implantación de dicha metodología a menudo exige un fuerte cambio cultural y organizativo en las organizaciones. Una correcta implantación de DevOps en una organización hará que los desarrolladores se despreocupen del entorno donde va a tener que correr su aplicación, proporcionándoles mucho más tiempo para dedicar a lo que realmente se les da bien, que es desarrollar; garantizando una mejor calidad del código, incrementando la productividad y en consecuencia disminuyendo los costes.

Uno de los pilares fundamentales del DevOps es el *feedback*. La retroalimentación entre ambos equipos es clave para conseguir la mejora continua, por ello es muy importante mantener reuniones periódicas y comunicación constante entre ambos equipos. También es de vital importancia la generación de informes que nos permitan analizar cómo están funcionando todos los procesos del proyecto. Existen numerosas herramientas de *reporting* que permiten realizar un seguimiento de este tipo de proyectos orientados al DevOps. Entre ellas, destacamos JIRA o Trello, software diseñado para supervisar todos los aspectos relacionados con el ciclo de vida de la aplicación.

Otro pilar del DevOps se basa en la automatización de procesos y evitar bloqueos por excesos de cargas de trabajo. Por ello, todo lo que pueda ser

candidato a ser automatizado, es preferible que se automatice. Es aquí donde nacen los servicios de Integración Continua y Entrega Continua como Jenkins, TravisCI o CircleCI. Los objetivos de estas herramientas consisten en agilizar y automatizar los procesos que conforman el núcleo del DevOps que consiste en la interacción entre ambos equipos para integrar código, testear el código y desplegar a producción.

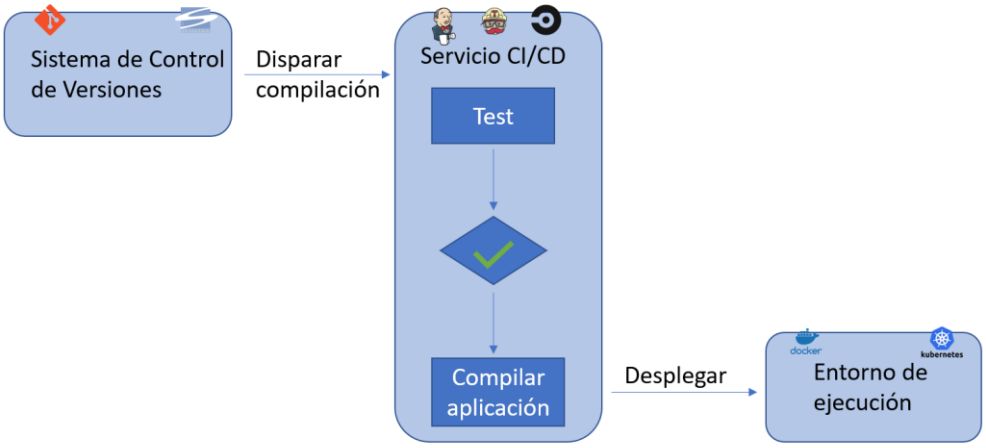


Figura 2. Diagrama de flujo de trabajo para DevOps.

Mediante estas herramientas, el equipo de desarrollo se limita a subir cambios a un repositorio manejado por un sistema de control de versiones. Al agregar un nuevo cambio se debe disparar un proceso de compilación y ejecución de la nueva versión en los entornos que hayan definido el equipo de sistemas. Una vez que la aplicación ya está ejecutándose en un entorno de pruebas, se lanzarán los tests que ha preparado el equipo de desarrollo. Por último, si los tests han pasado, se compila la aplicación de nuevo en un entorno similar al de pruebas que hayan definido en el equipo de operaciones para publicarse como nueva versión en producción.

CAPÍTULO 2. PRESENTACIÓN DE UN CASO DE USO Y SELECCIÓN DE TECNOLOGÍAS

Con la finalidad de aplicar las técnicas y herramientas que están siendo objeto de estudio, vamos a tomar una aplicación web que estamos desarrollando paralelamente y la usaremos para comparar y poner en práctica lo estudiado.

2.1. Presentación de la aplicación

Se trata de un producto para una empresa, la cual solicita una aplicación que proporciona la funcionalidad básica de un portal web para los empleados de la organización. Dicha aplicación deberá ser accesible públicamente desde Internet, aunque para acceder a los servicios que proporcione la misma sea necesaria previa autenticación.

La aplicación deberá proveer funcionalidades comunes a todos los empleados tales como: consulta de nómina, contrato, calendario laboral, diplomas de cursos de formación e información personal; rectificación de datos personales; solicitud de permiso de ausencia justificada; crear sugerencias anónimas; buzón de correo para poder comunicarse con el resto de los empleados.

Así mismo, la aplicación contará también con un sistema de roles que nos permita diferenciar un usuario normal de un usuario perteneciente al departamento de Recursos Humanos que será el encargado de administrar el portal. En consecuencia, contaremos con un rol de Personal de Recursos Humanos que podrá: dar de alta o baja un empleado para permitir o bloquear el acceso del usuario; visualizar, aceptar y rechazar solicitudes de rectificación de información personal; visualización de sugerencias anónimas.

En cuanto a requisitos no funcionales de la aplicación, el cliente solicita trabajar con dos versiones distintas de la aplicación a la misma vez. De forma que los nuevos cambios que se vayan integrando a lo largo del ciclo de vida de la aplicación, serán testeados durante un tiempo por el departamento de recursos humanos, conviviendo a la misma vez con la versión estable que utilizarán el resto de los usuarios. La nueva versión, tras ser utilizada y validada por los empleados de recursos humanos durante un periodo de tiempo, finalmente acabará sustituyendo a la versión antigua.

2.2. Análisis de tecnologías de desarrollo

Una vez que contamos con los requisitos de la aplicación, vamos a analizar las tecnologías involucradas en el desarrollo y funcionamiento de la aplicación, tales como: lenguaje de programación, sistema de persistencia, entorno de desarrollo.

Antes que nada, vamos a analizar la plataforma para la que vamos a desarrollar el sistema. Uno de los requisitos de los que disponemos es que la

aplicación debe ser accesible desde Internet. Esto nos obliga a desarrollar posiblemente una página web o una aplicación para móvil. Sin embargo, el personal de recursos humanos deberá administrar el portal diariamente desde su puesto de trabajo, con lo cual, por simple usabilidad vamos a priorizar la plataforma web por encima de la plataforma móvil.

2.2.1. Lenguajes de programación

Ahora que tenemos clara la plataforma para la que vamos a desarrollar la aplicación, vamos a ver qué lenguaje o lenguajes de programación utilizaremos para su desarrollo.

Vamos a tomar dos lenguajes candidatos y los evaluaremos en distintos aspectos con el fin de escoger la mejor alternativa.

Los lenguajes escogidos son Ruby y Node.js. La razón de esta elección viene dada, en primer lugar, porque nos gustaría conocer y aprender desarrollo web en Ruby, y en segundo lugar, porque en el Máster el estudiante ha cursado una asignatura (Tecnologías de navegador para las aplicaciones web) donde se imparte programación avanzada con Javascript y programación con Node.js.

Dicho esto, vamos a realizar valoraciones generales de ambos lenguajes.

Por una parte, de Ruby conocemos que tiene fama de ser un lenguaje muy flexible y amigable para el programador pues su sintaxis es altamente expresiva. Sin embargo, particularmente en nuestro caso va a requerir una mayor curva de aprendizaje puesto que es un lenguaje que no conocemos.

Por otra parte, tenemos Node.js que nos permite ejecutar código Javascript en el lado del servidor. Javascript es actualmente el lenguaje más utilizado para la web. Es un lenguaje altamente flexible y lo tenemos muy trabajado tanto en el Grado como en el Máster.

A continuación, vamos a evaluar las capacidades de ambos lenguajes en los siguientes ámbitos.

Grado de madurez y actualizaciones. Javascript [7] se lanzó en 1997 (ECMAScript 1) y desde entonces han aparecido cinco versiones más, hasta ECMAScript 2018, que fue liberada en 2018. Por su parte, Ruby [8] apareció en 1995 y ha sido revisado en numerosas ocasiones hasta la versión actual 2.7.0 liberada en 2019. En este aspecto Ruby parece tener un equipo de desarrollo más constante pese a que ambos tienen casi la misma madurez.

Comunidad y documentación. Pese a que Ruby tiene fama de contar con una amplia comunidad de calidad, Javascript no deja de ser el lenguaje más utilizado actualmente en el desarrollo web, con una extensa documentación y grandes comunidades que ofrecen soporte para este lenguaje.

Gestión de dependencias. Node.js utiliza npm, un gestor de dependencias muy potente cuyo uso está más que reconocido a lo largo de toda la comunidad de desarrollo. Npm es un gestor de paquetes que permite al desarrollador

integrar dependencias en sus proyectos de manera muy sencilla respetando la compatibilidad de versiones entre dependencias. Ruby por su parte tiene RubyGems, un gestor de paquetes, también denominados gemas, que permite al usuario agregar dependencias en sus proyectos.

***Frameworks*.** Ambos lenguajes son compatibles con variedad de *frameworks* de desarrollo web. Para Node.js disponemos de Express.js, Socket.io, Koa.js entre otros. Para Ruby tenemos RubyOnRails, Sinatra, Padrino, etc.

Compatibilidad con el resto del ecosistema. Ambos lenguajes tienen facilidad para integrarse con otros artefactos que compongan el sistema, como por ejemplo bases de datos relacionales (MySQL, Microsoft SQL Server, Oracle, PostgreSQL, etc.). Ambas tecnologías son de *backend*, pero además, sus distintos *frameworks* soportan una amplia variedad de motores de plantillas muy útiles para desarrollar el *frontend*; para Ruby destaca por encima del resto el motor de plantillas Slim, y sin embargo para Node.js tenemos varios motores muy utilizados como por ejemplo Handlebars o Jade. Tanto Node.js como Ruby son lenguajes para *backend*. No obstante, puesto que Node.js interpreta Javascript, nos capacita para desarrollar tanto *backend* como *frontend* con el mismo lenguaje, lo que nos garantiza una perfecta conexión entre ambas partes.

Concienciación con buenas prácticas de desarrollo. En este punto debemos destacar a Ruby, cuyas bases se fundamentan en las buenas prácticas de desarrollo. Fijándonos en el uso de Rails, el *framework* más usado para desarrollo web con Ruby, vemos que es extremadamente estricto con el uso de un determinado patrón de desarrollo, maneja las migraciones y cambios en el diseño de base de datos, genera un árbol de ficheros para el proyecto que casi nos obligan a seguir las prácticas que proponen, entre otras restricciones. En definitiva, resulta difícil no adaptarse a la metodología que propone. Por otro lado, Node.js es más flexible en este aspecto, dejando en manos del desarrollador la organización del proyecto.

Rendimiento y escalabilidad. En cuanto a rendimiento podemos decir que Node.js es mucho más rápido que Ruby en casi todos los aspectos, aunque a una escala tan pequeña como nuestra aplicación, que no va a tener mucho volumen de peticiones, puede ser que no lleguemos a apreciar la diferencia. No obstante, cabe destacar la programación asíncrona que utilizan las librerías subyacentes (en especial la librería encargada de E/S) que conforman Node.js, pues proporcionan un alto rendimiento sin bloqueos y una excelente experiencia de usuario, frente a la concurrencia bloqueante que podemos experimentar en Ruby. En cuanto a escalabilidad, por un lado tenemos a Ruby, que es ideal para grandes proyectos con equipo de numerosos desarrolladores, pues proporciona una filosofía de desarrollo que obliga a los programadores a mantener una disciplina; por otro lado tenemos la flexibilidad que ofrece Node.js que puede hacer que los desarrolladores se desencaminen del patrón de desarrollo a seguir. Hablando de rendimiento con respecto a la escalabilidad tenemos que hablar sobre los *Workers* y *Clusters* de Node.js que son capaces de crear y cerrar procesos a medida que la carga de trabajo va variando. Esto hace que las

aplicaciones desarrolladas en Node.js escalen muy bien en comparación con Ruby, cuya eficiencia se ve mermada en mayor nivel conforme la aplicación escala.

Por los factores estudiados anteriormente, decidimos escoger Node.js como tecnología para la programación de la aplicación web. Los principales motivos son: el estudiante posee experiencia avanzada de Javascript; mismo lenguaje para *backend* y *frontend*; mejor rendimiento y mayor flexibilidad.

2.2.2. Sistema de persistencia

A continuación, vamos a seleccionar la tecnología que vamos a utilizar para el almacenamiento de la información que requiere el sistema.

El caso de uso implica la transferencia esporádica de información de la base de datos de la empresa donde residen los datos de los empleados que son registrados y utilizados por el sistema software de planificación de recursos empresariales (ERP). Este programa actualmente conecta con una base de datos Microsoft SQL Server alojada en el CPD de la organización.

Con el fin de facilitar la integración de la información proveniente de la base de datos de la organización a la base de datos de la aplicación, decidimos que ésta también sea Microsoft SQL Server.

2.3. Integración Continua

Ya hemos elegido las tecnologías a utilizar para el desarrollo de la aplicación. Ahora vamos a escoger las estrategias y herramientas que conciernen a la fase de Integración Continua.

En primer lugar, vamos a necesitar una herramienta para el control de versiones. En este caso vamos a utilizar Git, puesto que es el sistema con el que más hemos trabajado. También es el más usado a nivel global y nos va a garantizar compatibilidad con el resto del ecosistema.

Como plataforma para almacenamiento de código utilizaremos GitHub en la nube, que es automáticamente integrable con Git y gratuita para repositorios de proyectos públicos.

En cuanto a la gestión de ramas y versiones del código, vamos a innovar y seguir una estrategia menos común. Utilizaremos dos ramas paralelas; una de ellas será la principal y la otra será la secundaria. El objetivo de esta estrategia es facilitar el cumplimiento del requisito de coexistencia de dos versiones distintas en producción.

Recordemos una de las estrategias de Despliegue Continuo denominada *Blue/Green*, en la cual teníamos una versión antigua en producción (*Blue*). Acto seguido desplegábamos la nueva aplicación en otro nodo (*Green*) y mediante un balanceador de carga redirigíamos las peticiones a la máquina con la nueva versión, liberando así el nodo con la versión antigua. Nuestro caso es parecido,

pero no llega a ser idéntico, porque nosotros sí que mantendremos el flujo de peticiones a ambos nodos. Aparte, vamos a incorporar una característica de la estrategia de despliegue denominada *Canary*, en la que un grupo de usuarios minoritario, y en este caso expertos, utilizaban y testeaban una nueva versión mientras que el resto de usuarios seguía con la versión antigua.

Dispondremos de dos ramas. Una rama principal (*Blue*) y otra rama secundaria (*Green*). La rama principal contendrá el código de la versión estable en producción donde accederán los usuarios normales. La rama secundaria será una copia de la rama principal sobre la que iremos implementando y liberando los nuevos cambios que el cliente vaya solicitando. Estos nuevos cambios solo podrán ser vistos por los empleados de recursos humanos, hasta que éstos validen los cambios y finalmente se publiquen en la rama *Blue.*

El modo de trabajar con cada rama se detalla a continuación.

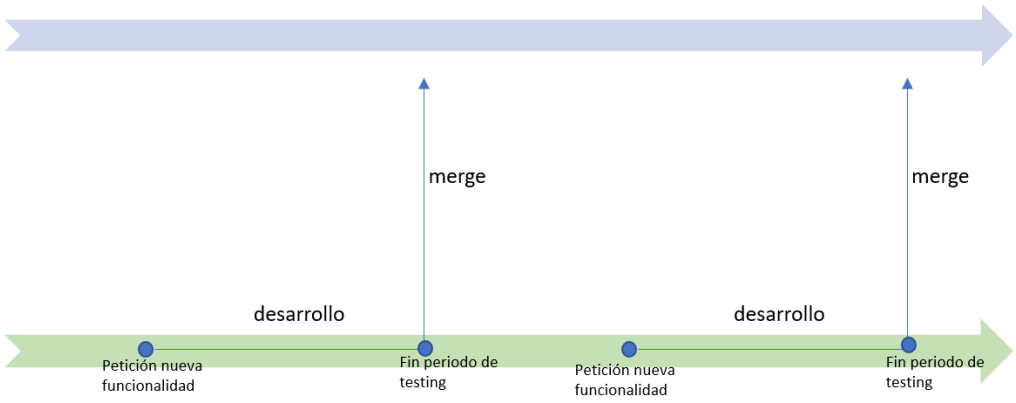


Figura 3. Diagrama de flujo de trabajo con Git.

Partimos de un inicio donde tenemos ambas ramas con el mismo código fuente de la aplicación. El cliente solicitaría que integremos una nueva funcionalidad a la aplicación. Descargaríamos el contenido de la rama *Green* a la rama *Blue* y comenzaríamos a trabajar en la nueva funcionalidad. Los cambios en la rama secundaria se publicarían y serían solo accesibles por los usuarios pertenecientes al grupo de recursos humanos, mientras que los usuarios normales seguirían visualizando la versión antigua. Una vez que estos usuarios expertos ya hayan validado la nueva funcionalidad, actualizaríamos el código de la rama *Blue* y desplegaríamos dicha rama en su nodo correspondiente; ahora ambos grupos visualizarían la misma versión de la aplicación e incluso podríamos apagar el nodo que aloja la rama *Green* y redirigir ambos grupos de usuarios al nodo *Blue* hasta que el cliente vuelva a solicitar un nuevo cambio.

2.4. Entrega Continua

Vamos a analizar qué tecnologías vamos a utilizar en la fase de Entrega Continua. En esta fase vamos a tener que lanzar pruebas sobre una aplicación, así que necesitaremos definir qué tipos de pruebas vamos a desarrollar y con

qué bibliotecas. También abordaremos las tecnologías que utilizaremos para el lanzamiento y gestión de los entornos necesarios para la ejecución de las pruebas.

De todas las tipologías de tests que hemos visto en el capítulo anterior vamos a escoger los tests de integración y los tests E2E.

2.4.1. Tests de integración

Para desarrollar tests de integración en Node.js existen varios *frameworks*. Entre los más destacados tenemos Jasmine, Mocha y Jest [9].

Jasmine. Es uno de los *frameworks* de *testing* más populares para Javascript. Integra casi todo tipo de estructuras necesarias para realizar tests, por lo que no tendremos que preocuparnos de instalar más dependencias para conseguir una completa funcionalidad. Por contra, tenemos el inconveniente de que abusa de la definición de variables globales.

Mocha. El *framework* para *testing* en Javascript más popular y utilizado. Destaca por su flexibilidad. Mocha define una estructura base de tests pero carece del resto de bibliotecas necesarias como por ejemplo de *assertions*, *mocks*, etc., obligando al desarrollador a descargar bibliotecas que provean estas estructuras. Por una parte, puede parecer un inconveniente porque tienes que descargar dependencias adicionales para realizar un proyecto de *testing* completo; lo que por ejemplo con Jasmine no pasaría porque integra todas las estructuras necesarias. Pero si buscamos la parte positiva, podemos pensar que, si no vamos a necesitar por ejemplo los *mocks*, es espacio y recursos que estamos ganando al no tener que incluir estructuras innecesarias que no vamos a emplear. Además, está ofreciendo la posibilidad de utilizar las bibliotecas de estructuras que queramos, lo que garantiza mayor flexibilidad.

Jest. Recomendado por los desarrolladores de Facebook. Implementa una característica denominada "*testing* en paralelo" pues es capaz de paralelizar la ejecución de tests y ahorrar mucho tiempo sobre todo en proyectos de gran envergadura. Quizás demasiado potencial como para nuestra pequeña aplicación. Hace uso de variables globales como Jasmine, lo que es un factor en contra.

Vamos a quedarnos con **Mocha** por la flexibilidad que nos ofrece, porque no usa variables globales y porque como solamente vamos a desarrollar tests de integración con esta biblioteca, únicamente necesitaremos estructuras de *assertions*.

Puesto que vamos a utilizar Mocha, y necesitaremos una biblioteca de *assertions*, descargaremos la recomendada por el propio autor: **Chai**. Chai es una biblioteca de *assertions* enfocada al desarrollo basado en pruebas (TDD), muy ligera y fácilmente integrable con Mocha.

2.4.2. Tests E2E

Tenemos varias opciones disponibles para desarrollar esta tipología de tests. Entre las más populares: CasperJS, Protractor y CodeceptJS [10]. A la hora de elegir un *framework* para *testing* E2E es importante tener en cuenta la compatibilidad de dicho *framework* con *driver*s para navegador, porque estos tests necesitan disponer de un navegador en ejecución sobre el que lanzar acciones.

CasperJS. No es una solución nativa para Node.js sino que fue desarrollada en Python para *testing* en esa misma plataforma. Sin embargo, han desarrollado un *wrapper* para Node.js que está dando muy buenos resultados y es fácilmente integrable utilizando el gestor de paquetes npm. Está preparado para correr con motores de navegadores artificiales como por ejemplo PhantomJS [11] o SlimerJS que son simuladores de navegadores cuyo fin es elevar el rendimiento y reducir el tiempo de ejecución de los tests. La sintaxis necesaria para desarrollar los tests es algo compleja.

Protractor. Un *framework* muy popular. El más usado en *testing* para aplicaciones desarrolladas en Angular. Está muy enfocado en sintaxis Angular. A diferencia que CasperJS, Protractor trabaja solo con navegadores reales.

CodeceptJS. Es más que un *framework*. Provee un *wrapper* o interfaz que permite elegir el *framework* que queramos dentro de un conjunto soportado. Entre los *frameworks* que soporta: *Webdriver*IO, Protactor, Selenium *Webdriver* JS, NightmareJS. La sintaxis es extremadamente sencilla de programar y entender. Una gran ventaja de la que dispone es que podemos cambiar la implementación de *framework* cambiando un solo parámetro de la configuración. Igual que Protractor, solo trabaja con navegadores reales.

Nos quedaremos con CodeceptJS por la gran flexibilidad que ofrece al actuar de *wrapper* de distintos *frameworks* y por la sencillez de su sintaxis que hace que tengamos que dedicar poco tiempo al aprendizaje del API.

Ahora que tenemos un *framework* de desarrollo de tests E2E, vamos a escoger un *webdriver* compatible. Uno de los más utilizados en el mundo del *testing* E2E es el conocido Selenium *Webdriver* [12]. Estos *driver*s consisten en una API que escucha en un determinado puerto. Al ejecutar un test, éste envía un mensaje a la API para que ejecute una acción sobre el navegador compatible que está en ejecución. Selenium *Webdriver* proporciona compatibilidad con los navegadores más comunes, en nuestro caso lo utilizaremos con Firefox.

2.4.3. Entorno de ejecución

Una vez que ya tenemos claro los tipos de tests que vamos a ejecutar, vamos a decidir la arquitectura del entorno de ejecución y las tecnologías que vamos a emplear para ejecutar dichos entornos.

Cuando en el capítulo anterior veíamos las distintas alternativas para crear entornos de ejecución distinguíamos entre tres posibilidades: máquinas físicas, máquinas virtuales y contenedores.

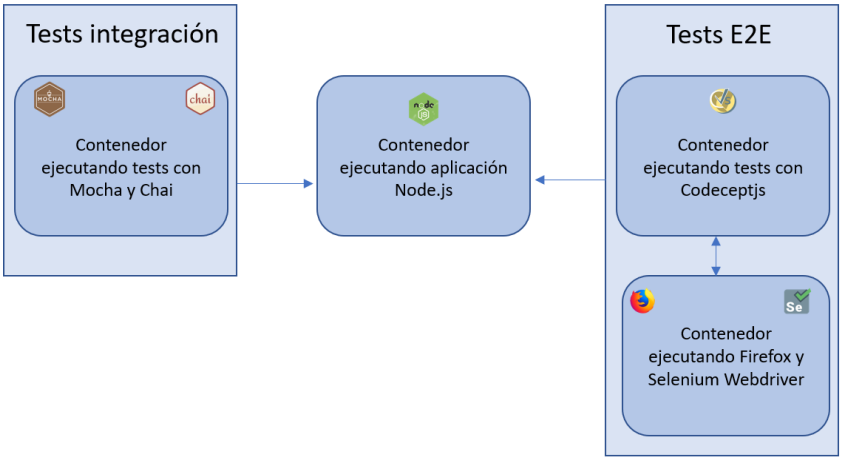


Figura 4. Esquema de funcionamiento y comunicación entre los nodos encargados del *testing.*

La estrategia que vamos a seguir es la de crear una máquina donde se ejecute la nueva versión de la aplicación. Una vez tenemos la aplicación lanzada en un entorno, ejecutaremos los tests de integración sobre dicho entorno; para ello lanzaremos una nueva máquina que ejecutará los tests contra la primera. Por último, ejecutaremos los tests E2E; en este caso vamos a lanzar dos máquinas, una que se encargue de ejecutar los tests contra la máquina que ejecuta la aplicación, y otra que se encargue de ejecutar un navegador y un *webdriver*. De este modo nuestra máquina de *testing* accederá a las dos máquinas para validar las pruebas

Para crear los entornos de ejecución hemos decidido optar por la virtualización. Comparando Vagrant y Docker para la generación de entornos de ejecución vemos que ambas herramientas son mucho más flexibles que una máquina física o una máquina virtual tradicional.

A continuación, presentamos una comparativa entre Docker y Vagrant atendiendo a los aspectos que más nos interesan.

Característica	**Docker**	**Vagrant**
Tiempo de arranque	< 10 segundos	minutos
Consumo de recursos	Muy bajo	Alto
Facilidad de replicación y escalabilidad	Muy fácil	No es trivial y es poco eficiente
Tiempo de apagado	segundos	segundos
Espacio en disco necesario	< 100 mb	> 1 gb

Por los motivos que refleja la tabla anterior decidimos decantarnos por el uso de Docker como proveedor de entornos de ejecución.

2.5. Despliegue Continuo

En esta fase se va a llevar a cabo la puesta en producción de la aplicación. A continuación, veremos la estrategia de despliegue que seguiremos y las herramientas para la gestión y creación del entorno donde residirá la aplicación final.

Puesto que hemos utilizado una solución orientada a contenedores en la fase anterior, para el despliegue también utilizaremos Docker. A priori, utilizar un contenedor como entorno de ejecución para nuestra aplicación de producción puede parecer algo inseguro o poco robusto, sin embargo, es una práctica que se está llevando a cabo continuamente en grandes compañías como Twitter, Google o Amazon.

Además, no buscamos un entorno robusto con extensas configuraciones y muchos servicios en funcionamiento, lo que buscamos son entornos de ejecución ligeros, aislados, suficientes para alojar nuestra aplicación y que podamos lanzarlos y apagarlos con facilidad y agilidad, permitiéndonos una alta tasa de actualizaciones de versiones de aplicación minimizando las caídas de servicio.

Teniendo en cuenta el requisito que hace referencia a la necesidad de publicar dos versiones de aplicación distintas en algún momento, está claro que vamos a necesitar levantar como mínimo dos contenedores cuando se produzca este caso.

Recordemos que cada vez que vayamos a integrar un cambio en la aplicación deberemos publicarlo para un grupo reducido de usuarios que actuarán de *testers* de la aplicación; una vez esos cambios sean aceptados por los usuarios expertos, la nueva versión pasará a ser accesible para todos los demás usuarios también.

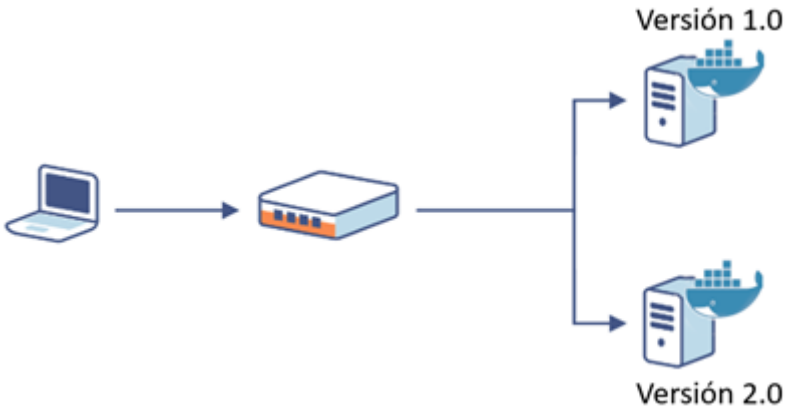


Figura 5. Diagrama de funcionamiento del balanceador de carga.

Entonces sabiendo que necesitamos dos contenedores levantados necesitaremos también un artefacto que se sitúe por encima de dichos contenedores y sea capaz de discriminar la petición entrante para servirla a un contenedor u otro. Esta funcionalidad nos la podrá proporcionar un balanceador de carga.

Un balanceador de carga es un dispositivo hardware o software que se pone delante de un conjunto de servidores que tienen desplegada una o varias aplicaciones. Este dispositivo distribuye las solicitudes que llegan de los clientes a los distintos puntos de entrada de la aplicación. Esta distribución o balanceo se suele realizar usando algún algoritmo que garantice una asignación equitativa de carga de trabajo a cada nodo final. Sin embargo, también podemos implementar otro tipo de reglas en el balanceador para hacer que la distribución de peticiones sea más sofisticada que lo que sería la utilización de un algoritmo de distribución; esta funcionalidad es la que estamos buscando.

Existen numerosas alternativas en cuanto a balanceadores de carga: Nginx, Traefik, HAProxy. Más allá de qué proveedor utilicemos para el balanceo, necesitamos saber en qué capa situar dicho balanceador. La implementación de reglas o lógica para el balanceo de una petición basándonos en el contenido de la propia petición se puede establecer a distintos niveles o capas: capa 4 o capa 7. El balanceo de capa 4 se utiliza para redirigir el tráfico TCP, normalmente el balanceo a este nivel se utiliza para redistribuir el volumen de peticiones y ganar en tiempo de respuesta en aplicaciones con mucha sobrecarga de clientes. El balanceo en capa 7 redirige el tráfico a nivel de capa de aplicación (HTTP/HTTPS), admite redireccionamiento basado en rutas y puede ser capaz de leer las cabeceras de las peticiones.

Nosotros vamos a realizar el balanceo en capa 7 y utilizaremos algún tipo de marcador en la cabecera de la petición para que el balanceador sea capaz de discriminar a qué contenedor redirigir la petición.

Como proveedor usaremos HAProxy porque ya tenemos experiencia tras su estudio en una de las asignaturas del Máster (Computación en la nube y soluciones de virtualización).

2.6. DevOps

Puesto que nuestro equipo está formado una única persona, no tiene sentido integrar al completo un sistema de DevOps, pues no hay manera de demostrar la correcta integración de ambos equipos. Así que descartamos el uso de herramientas de *reporting* y *feedback* cuyo fin es mejorar la comunicación e integración del equipo de desarrollo y el de sistemas.

No obstante, sí que vamos a desplegar un servicio de Integración Continua y Entrega Continua puesto que esto sí que es viable. Para ello vamos a elegir la opción que más nos pueda favorecer en nuestro caso de uso.

Las alternativas de servicios de Integración y Entrega Continua con las que contamos son las siguientes:

TravisCI [13]**.** Se encuentra alojada en la nube y es compatible con repositorios de código alojados en la nube mediante GitHub. Es gratuito para repositorios públicos, pero requiere comprar licencia de uso para utilizarlo con repositorios privados. Para configurar un flujo de fases de Entrega Continua se utiliza un fichero llamado *.travis.yml* que debe residir en el directorio raíz de nuestro repositorio. Este fichero especifica el lenguaje de programación del proyecto, dependencias necesarias, y configuraciones de entornos para compilar y ejecutar nuestra aplicación. Es compatible con Docker y con las plataformas *IaaS* más conocidas para la fase de despliegue: AWS, Azure, Heroku.

CircleCI [14]**.** Al igual que Travis se encuentra alojado en la nube. Económicamente más asequible que Travis. Configura el flujo de trabajo mediante un fichero YAML.

Jenkins [15]**.** A diferencia que los anteriores servicios, Jenkins es un software autocontenido de Integración y Entrega Continua y no se ejecuta en la nube sino que es un software descargable que necesita un servidor sobre el que ejecutarse. Es con diferencia el que más tiempo lleva asentado en el mercado y el más completo y personalizable. Cuenta con más de 1000 *plugins* que extienden su funcionalidad y es compatible con la mayoría de las herramientas que más se utilizan.

Tras analizar brevemente las distintas herramientas, nos quedamos con Jenkins porque nos garantiza una alta compatibilidad con muchas herramientas, por ser código abierto y por la capacidad de personalización.

CAPÍTULO 3. IMPLEMENTACIÓN DE LA APLICACIÓN

En este capítulo veremos en detalle los aspectos relacionados con la implementación de la aplicación y la configuración de las herramientas necesarias para desplegar la plataforma de Integración y Despliegue Continuo (CI/CD).

El código de la aplicación ya terminada se encuentra alojado en un repositorio GitHub de la cuenta del estudiante:

https://github.com/osmartinez/portal-empleado

Se ha realizado también el diseño de la base de datos:

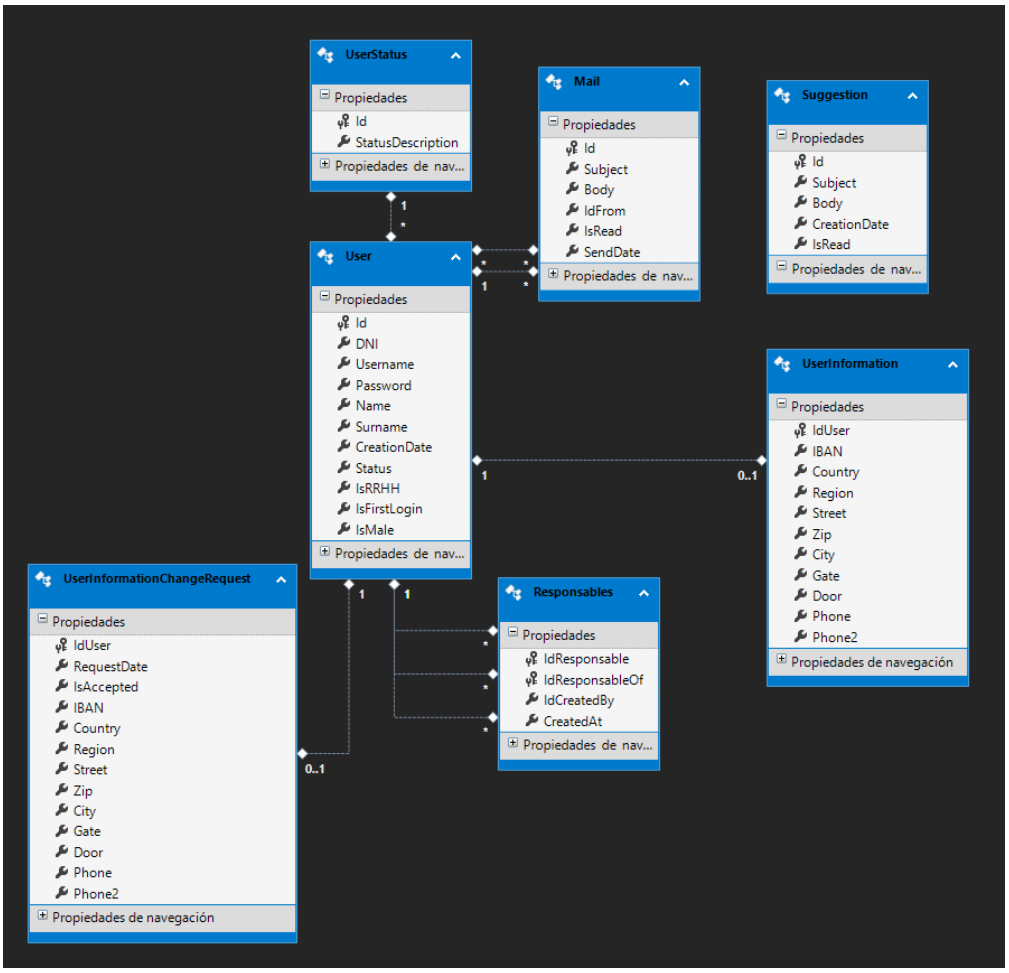


Figura 6. Esquema del diseño de base de datos.

Utilizaremos Azure como proveedor. Azure nos ofrece 170 euros para utilizar en su portal de servicios durante un mes con la finalidad de realizar todo tipo de pruebas.

Hemos decidido alojar la base de datos como un servicio externo. Este servicio que ofrece Azure se denomina Azure SQL Database y es un servicio administrado de base de datos relacional de uso general que ofrece un rendimiento escalable. Consideramos que es una buena elección contar con un servicio de persistencia externalizado por razones de mantenimiento y escalabilidad.

A continuación, vamos a ver la configuración de todo el ecosistema de herramientas y servicios necesarios para montar el sistema de Integración Continua.

3.1. Despliegue y configuración de servidor Jenkins

3.1.1. Creación de la instancia

En este apartado vamos a levantar una máquina virtual en la nube con Jenkins accesible públicamente que será el nodo encargado de realizar el proceso de Integración, Entrega y Despliegue Continuo.

Para realizar este proceso vamos a trabajar desde la consola de comandos que nos ofrece Azure; en nuestro caso accederemos a la consola desde el panel web del portal de servicios Azure.

Vamos a crear un fichero *cloud-init* de inicialización de una máquina. Los ficheros *cloud-init* permiten definir una serie de mandatos que se ejecutan en tiempo de arranque de una instancia.

```
touch cloud-init-jenkins.txt
nano cloud-init-jenkins.txt
```

Ahora vamos a editar el fichero con el siguiente contenido:

```
#cloud-config
package_upgrade: true
write_files:
  - path: /etc/systemd/system/docker.service.d/docker.conf
    content: |
      [Service]
        ExecStart=
        ExecStart=/usr/bin/dockerd
  - path: /etc/docker/daemon.json
    content: |
      {
        "hosts": ["fd://","tcp://127.0.0.1:2375"]
      }
runcmd:
  - apt install openjdk-8-jre-headless -y
  - wget -q -O - https://pkg.jenkins.io/debian/jenkins-ci.org.key | sudo apt-key add -
  - sh -c 'echo deb https://pkg.jenkins.io/debian-stable binary/ >
/etc/apt/sources.list.d/jenkins.list'
  - apt-get update && apt-get install jenkins -y
  - curl -sSL https://get.docker.com/ | sh
  - usermod -aG docker azureuser
```

```
    - usermod -aG docker jenkins
    - service jenkins restart
```

En el primer bloque estamos permitiendo la actualización de paquetes. En el segundo bloque estamos creando dos ficheros (docker.conf y daemon.json). En el último bloque ejecutamos mandatos en consola para instalar la openjdk 8, descargar Jenkins, descargar Docker, instalarlo, añade el usuario “azureuser” y el usuario “Jenkins” al grupo de usuarios “docker”. Por último, reinicia el servicio de Jenkins.

Ahora vamos a crear un grupo de recursos. Esto es un contenedor que almacena los recursos relacionados con una solución Azure.

```
az group create --name myResourceGroupJenkins --location westeurope
```

Ya hemos creado el contenedor de recursos. En este caso lo hemos localizado en Europa del Oeste. Si quisiéramos ver todas las localizaciones disponibles podríamos ejecutar en la consola:

```
azure location list
```

Creamos la máquina virtual especificando el grupo de recursos y el fichero *cloud-init* que hemos creado anteriormente:

```
az vm create --resource-group myResourceGroupJenkins \
  --name jenkinsMachine \
  --image UbuntuLTS \
  --admin-username azureuser \
  --generate-ssh-keys \
  --custom-data cloud-init-jenkins.txt
```

La máquina la hemos llamado “jenkinsMachine”, hemos escogido una imagen LTS de Ubuntu, le hemos indicado quién es el administrador de dicha máquina y obligamos a generar el par de claves ssh con el fin de poder acceder mediante ssh.

Cuando la máquina esté levantada, vamos a especificar dos directivas de seguridad de grupos para permitir el tráfico en los puertos 8080 para poder acceder al panel web de Jenkins y 3000 para habilitar el acceso a nuestra aplicación web cuando esté desplegada.

```
az vm open-port --resource-group myResourceGroupJenkins --name jenkinsMachine --
port 8080 --priority 1001
```

3.1.2. Configuración Jenkins

Vamos a visualizar la IP pública de la instancia que acabamos de crear. Esto lo podemos hacer navegando por el panel web accediendo a la configuración de red de nuestras máquinas virtuales o ejecutando el siguiente comando en la consola de Azure:

```
az vm show --resource-group myResourceGroupJenkins --name jenkinsMachine -d --
query [publicIps] --o tsv
```

Conectamos vía SSH por el puerto 22 a la máquina virtual de Jenkins. Podemos acceder desde la consola de azure sin necesidad de agregar el par de claves explícitamente o podríamos descargar las claves generadas e integrarlas en nuestro PC.

```
ssh azureuser@40.68.185.174
```

Ahora realizaremos una petición a la web http://40.68.185.174:8080 donde podremos ver levantado el servidor web con Jenkins. Al acceder nos pedirá una contraseña de inicialización que se encuentra en el fichero alojado en */var/lib/jenkins/secrets/initialAdminPassword.* Para iniciar sesión, necesitamos conocer el contenido de dicho fichero que podremos visualizar ejecutando:

```
azureuser@jenkinsMachine:~$ sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```

Una vez que hemos accedido a Jenkins por primera vez nos solicitará que indiquemos los plugins que va a instalar. De momento solo instalaremos el plugin de GitHub.

3.1.3. Generación de webhook

Un *webhook* es un disparador que se activa cuando realizamos una determinada acción sobre un repositorio de código. Este disparador puede desencadenar otra acción en el servidor de Integración Continua.

Para configurar un *webhook* en GitHub, vamos a necesitar navegar al repositorio GitHub de nuestra aplicación.

https://github.com/osmartinez/portal-empleado/settings/hooks

Desde dicho panel, agregaremos un nuevo *webhook*.

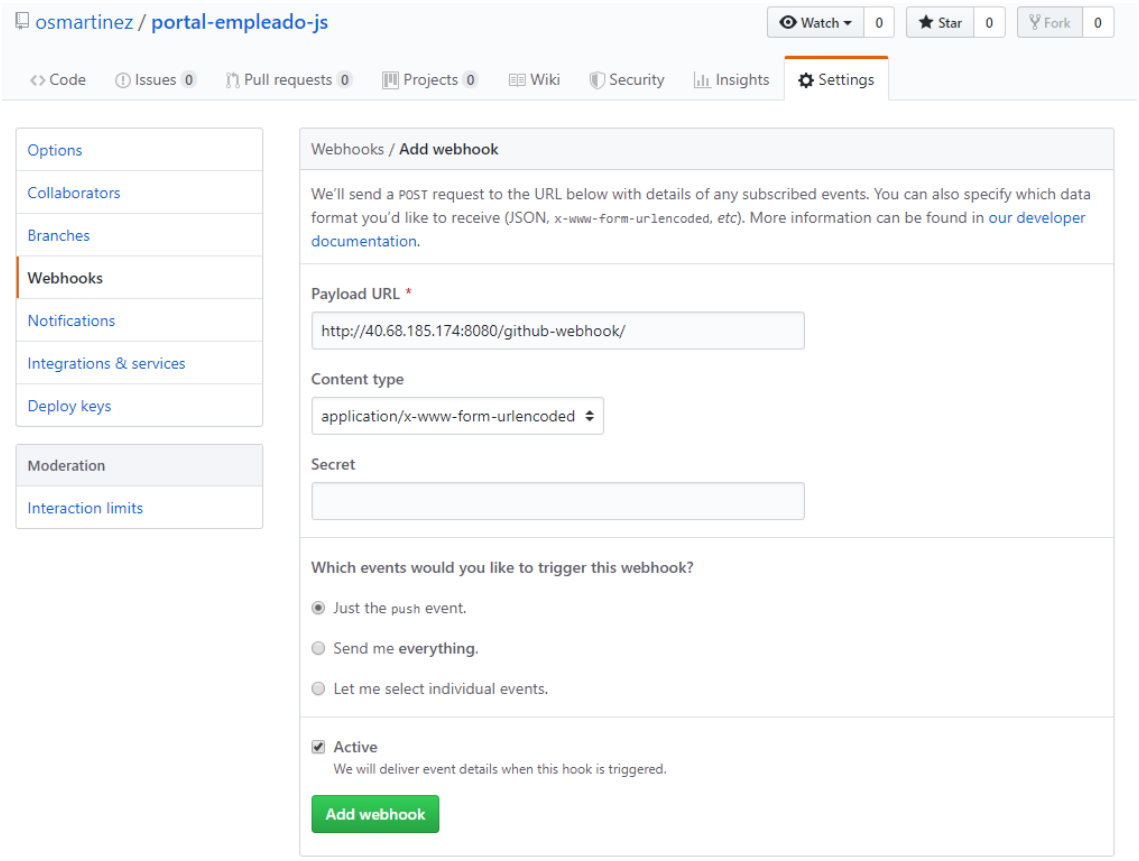


Figura 7. Creación de un *webhook* con Github.

Como se puede ver en la imagen, especificamos la URL donde GitHub debe realizar la petición para informar a nuestro servidor de integración de que un nuevo cambio ha sido agregado. También especificamos que solamente accionaremos el disparador cuando se realice un *push* al repositorio.

3.2. Descripción e implementación del pipeline. Paso a paso

Jenkins, el servidor CI/CD que estamos utilizando, proporciona una característica denominada Jenkins Pipeline. Esta característica está compuesta por un conjunto de *plugins* que nos permiten definir flujos de Entrega Continua para nuestros proyectos. Un flujo de Entrega Continua es una consecución automática de procesos que se ejecutan sin intervención humana y que tienen la finalidad de hacer llegar de manera rápida y segura al usuario final cualquier nuevo cambio en la aplicación implementado por el equipo de desarrollo. Estos flujos de Entrega Continua son totalmente personalizables pudiendo definirse desde Pipelines muy simples con pocas fases, hasta Pipelines altamente complejas, dependiendo del tamaño del proyecto y del número de etapas por las que queramos que circule nuestro código.

Las Pipelines al fin y al cabo son simples ficheros de texto denominados Jenkinsfile. Estos ficheros tienen una estructura básica muy fácil de entender a simple vista. A continuación, se presenta el esqueleto de un Jenkinsfile:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                //
            }
        }
        stage('Test') {
            steps {
                //
            }
        }
        stage('Deploy') {
            steps {
                //
            }
        }
    }
}
```

En primer lugar, tenemos la definición del agente (*agent*), muy útil cuando trabajamos con tecnologías de contenedores, como por ejemplo Docker o Kubernetes, donde podemos especificar el agente encargado de ejecutar la Pipeline. En general se declara un agente común para todas las fases (*stages*) pero también tenemos la posibilidad de definir un agente para cada fase.

Seguidamente vemos la definición de las fases que componen el flujo de Entrega Continua. Cada fase tiene un alias y podemos definir tantas fases como queramos.

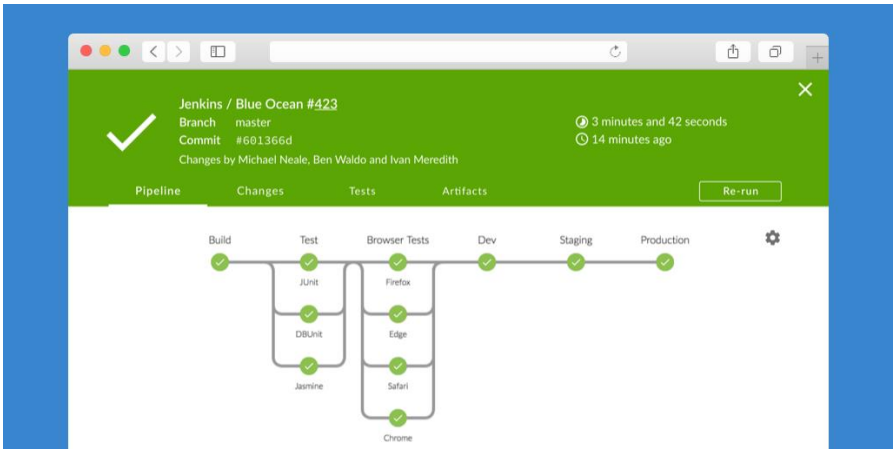
Dentro de cada fase se definen los pasos (*steps*) que se van a ejecutar en ese momento.

Figura 8. Ejemplo ejecución paralela de pipeline en fase de *testing*.

Es importante tener en cuenta que la ejecución de las fases de las que está compuesta una Pipeline se produce linealmente. Aunque tenemos la posibilidad de ejecutar varias fases en paralelo utilizando bloques *parallel,* muy útiles para la fase de *testing*, cuando por ejemplo tenemos que testear el código en distintas

plataformas; en estas situaciones nos puede interesar ejecutar procesos en paralelo para minimizar la duración y agilizar el despliegue.

Este fichero Jenkinsfile debe residir en el repositorio de nuestro proyecto. El hecho de que estos ficheros residan en nuestro repositorio enriquece enormemente esta característica, puesto que al ser un fichero más, es controlado también por nuestra herramienta de control de versiones. Jenkins proporciona una interfaz web para desarrollar estas Pipelines, sin embargo, al ser ficheros de texto, los programadores también pueden crear y editar Pipelines sin necesidad de acceder al panel de Jenkins.



Desde 2016, hay disponible un plugin para Jenkins denominado *Blue* Ocean [16], el cual proporciona una interfaz web renovada para definir Pipelines de manera mucho más intuitiva y amigable para el desarrollador. Nosotros utilizaremos este plugin para definir nuestras Pipelines.

En los siguientes apartados describimos el flujo de integración que va seguir el código de nuestra aplicación desde el momento en que mandemos subir el cambio al repositorio, hasta que el cambio es desplegado en producción a través de una nueva versión de aplicación.

Figura 9. Nuestra pipeline.

Tal y como se aprecia en la figura, nuestra Jenkins Pipeline se compone de ocho pasos. Los dos primeros (Start y Build) corresponden al proceso de Integración Continua; Test y TestE2E pertenecen al proceso de Entrega Continua, y el resto pertenecen a la fase de Despliegue Continuo.

3.3. Integración Continua

Comenzaremos por el sistema de control de versiones. Como ya sabemos, Git define una rama principal (*master*) para cada repositorio. Lo que haremos es prescindir de esta rama y crear dos nuevas ramas.

```
git checkout -b blue
git checkout -b green
```

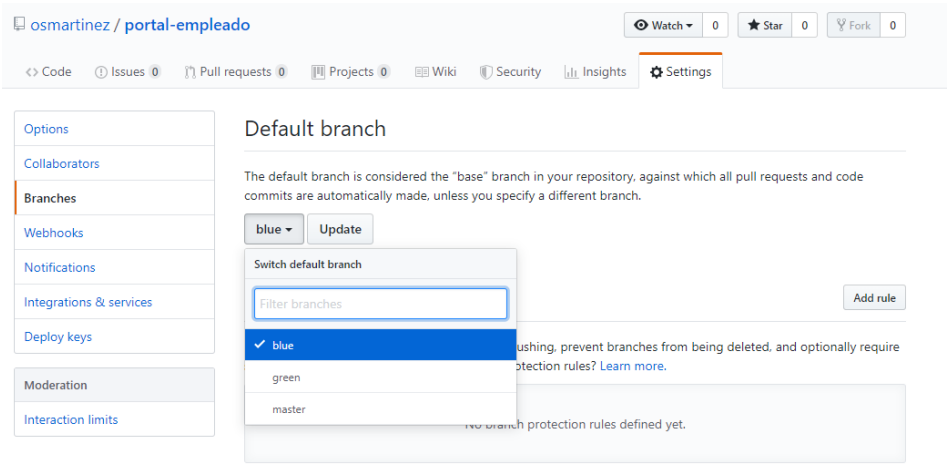


Figura 10. Sustitución de rama principal máster/blue en GitHub.

Con nuestras ramas ya creadas, accedemos al panel web de GitHub y deshabilitamos el uso de la rama *master* como rama principal. En su lugar asignaremos dicho rol a la rama *Blue*.

Ahora ya podemos desarrollar nuestra aplicación sobre la rama *Blue*.

Con esta estrategia lo que vamos a conseguir es tener continuamente diferenciadas y separadas dos versiones distintas de la aplicación, pudiendo en todo momento volcar el código de una a otra cuando queramos servir la misma versión a todos los usuarios.

Cada vez que decidamos subir un nuevo cambio a una de las ramas se propagará una petición a nuestro servidor Jenkins que hará que se comience a ejecutar la Pipeline de dicha rama. Pongamos un ejemplo con un nuevo cambio realizado en la rama *Green* (rama que sirve la aplicación al personal de recursos humanos):

```
git add .
git commit -m “Añade funcionalidad para cambiar el domicilio de un empleado”
git push origin green
```

Esto disparará la ejecución del Pipeline de la rama *Green*. En el primer nodo del proceso (*Start*) se descargarán los cambios del repositorio remoto que ha habido en la rama que provocó el *webhook.* Todo este proceso lo realiza

automáticamente Jenkins. Una vez que están todos los ficheros en el disco del servidor, si no ha habido ningún problema durante la descarga se ejecutará el siguiente paso (*Build*).

En la fase *Build* es en la que se realiza la compilación de la aplicación para ejecutar posteriormente la batería de pruebas sobre ella. En nuestro caso, vamos a utilizar Docker para generar un entorno de ejecución idéntico al de producción sobre el cual ejecutaremos parte de los tests de nuestra aplicación. Vamos a definir la imagen a partir de la cual el contenedor va a arrancar (Dockerfile):

```
FROM node:alpine

ARG PUERTO
ENV PUERTO ${PUERTO}

EXPOSE ${PUERTO}
WORKDIR /var/www/
COPY package.json /var/www/
RUN npm install
COPY src/ /var/www/
```

En este fichero estamos diciendo que utilizaremos una imagen de base “node:alpine”. También definimos un puerto que vamos a recibir como parámetro, que será el puerto a través del cual la aplicación será publicada. Establecemos el directorio de trabajo, copiamos el fichero “package.json” del repositorio descargado al directorio /var/www. Seguidamente ejecutamos “npm install” para descargar todas las dependencias necesarias para ejecutar la aplicación. Por último, terminamos copiando todo el código de la aplicación desde el repositorio descargado que se encuentra en la carpeta src/. Este Dockerfile va dentro del repositorio de la aplicación, así que desde la Pipeline, en la fase *Build* vamos a ejecutar los siguientes comandos:

```
docker stop portal-empleado && docker rm portal-empleado
docker build -t portal-empleado --build-arg PUERTO=5555 -f Dockerfile .
docker run --net mynet --name portal-empleado -p 5555:5555 portal-empleado node /var/www/index.js 5555 &
```

Primeramente, nos aseguramos de abortar la ejecución del propio contenedor que vamos a crear por si acaso se hubiera quedado abierto por algún tipo de fallo o bug.

Seguidamente construimos la imagen a partir del Dockerfile que definimos anteriormente. Especificamos que la aplicación se publique en el puerto 5555.

Como paso final lanzamos y publicamos el contenedor en el puerto 5555 y ejecutamos la orden “node /var/www/index.js 5555 &” que lo que hace es ejecutar nuestra aplicación Node.js en segundo plano.

Si no ha ocurrido ningún error durante el lanzamiento del contenedor, esta fase debería pasar en verde y comenzar la ejecución de la siguiente fase (Test).

3.4. Entrega Continua

Ya tenemos el código de la aplicación descargado en el volumen de almacenamiento de la máquina Jenkins y tenemos también un contenedor corriendo nuestra aplicación escuchando en el puerto 5555.

Figura 11. Ejemplo de cancelación de pipeline por fallo en fase de Test.

El objetivo las fases de test no es otro que abortar la ejecución de la Pipeline en caso de que el nuevo cambio en la aplicación no sea capaz de pasar las pruebas que hemos diseñado.

3.4.1. Tests de integración

Ahora vamos a generar un contenedor que lanzará las pruebas contra el contenedor que está corriendo la aplicación. Para ello vamos a definir otro Dockerfile que llamaremos “Dockerfile.test”, con el siguiente contenido:

```
FROM portal-empleado

RUN mkdir /var/test
WORKDIR /var/test
COPY package.json /var/test/
RUN npm install && npm install -g Mocha
COPY src/ /var/test/
CMD ["Mocha", "tests/unitTests/", "--recursive", ";", "Mocha", "--exit"]
```

En este fichero especificamos que utilice como imagen base la que ya hemos generado en la fase anterior y que denominamos “portal-empleado”. Seguidamente descargamos las dependencias e instalamos Mocha a nivel global para poder lanzar los tests. Cuando el contendor se encuentre levantado, ejecutaremos un mandato (CMD) que comenzará la ejecución de todos los tests funcionales de la aplicación localizados en la carpeta tests/unitTests/.

Desde la Pipeline simplemente construimos la imagen y lanzamos el contenedor:

```
docker build -t portal-empleado-test -f Dockerfile.test .
docker run  --rm portal-empleado-test
```

Las pruebas comenzarán a ejecutarse al acabar de levantarse el contenedor. Estos tests se han desarrollado utilizando la librería Mocha para Node.js. Con estos tests validaremos principalmente que cada tipo de usuario puede acceder

a las páginas que le corresponden. En esencia testearemos los permisos de acceso a páginas.

Veamos un ejemplo de test de integración desarrollado con Mocha. En dicho test se comprueba que un visitante, sin haberse autenticado, no puede acceder a ningún recurso interno del sistema ni página web que no sea la de login:

```
    describe('Probando permisos visitante', function (done) {
        it('Al intentar acceder al panel de RRHH debería redireccionar a la página dashboard',
(done) => {
            request(app).get('/rrhh')
                .expect('Location', '/dashboard')
                .expect(302, done);
        });
        it('Al intentar acceder al home debería ser redirigido al login panel', (done) => {
            request(app).get('/dashboard')
                .expect('Location', '/auth/login')
                .expect(302, done);
        });
    });
```

Como podemos ver en el test anterior, se le da mucha importancia a la descripción del test. Existen dos funciones “describe” y “it”. “describe” declara una descripción para una batería de tests sobre una misma temática. Dentro de dicha función se ejecutan funciones “it” que detallan en lenguaje natural el caso de test y el resultado esperado; posteriormente ejecuta el test. Esta manera de definir descripciones para conjuntos de tests y casos de tests es muy útil a la hora de visualizar los reportes que se generan durante la ejecución de los tests, pues facilitan muchísimo la tarea de localización de tests fallidos.

A continuación visualizamos un fragmento de la salida obtenida durante la ejecución de los tests anteriormente mostrados:

```
    Probando permisos visitante
    GET /rrhh 302 0.944 ms - 32
      ✓ Debería redireccionar a la página dashboard
    GET /dashboard 302 0.391 ms - 33
      ✓ Debería redireccionar a la página de acceso
```

3.4.2. Tests E2E

La ejecución de tests E2E viene a ser una tarea más tediosa que la de ejecución de otros tipos de tests. La complejidad radica en la necesidad de contar con un navegador web, y más aún si contamos con tecnologías de contenedores para ejecutar nuestros tests. Recordemos que este tipo de tests emulan el comportamiento de un usuario humano utilizando la aplicación.

Como primer paso vamos a lanzar un contenedor auxiliar que va a ejecutar un navegador y un *webdriver*. Un *webdriver* es una API a través de la cual podemos ejecutar acciones sobre un navegador mediante código. Uno de los *webdriver* más utilizados y con mayor aceptación es Selenium *Webdriver*, que

además, proporciona *driver*s para comunicarnos con los navegadores más utilizados (Firefox, Chrome, IE, etc.).

Para lograr este paso seguiremos alguna de las indicaciones que se proponen en la documentación del repositorio:

https://github.com/SeleniumHQ/docker-selenium

En nuestro Pipeline, en la fase Test E2E vamos a ejecutar estos dos mandatos:

```
docker stop firefox-container && docker rm firefox-container
docker run -d -p 4444:4444 --net mynet -v /dev/shm:/dev/shm --name firefox-container selenium/standalone-firefox:3.12.0-americium
```

Con estas instrucciones lanzamos un contenedor con un navegador Firefox y un *webdriver* de Selenium escuchando en el puerto 4444 para ejecutar acciones sobre el navegador.

Como segundo paso vamos a necesitar otro contenedor que sea capaz de ejecutar los tests desarrollados sobre la aplicación desplegada en el contenedor que lanzamos en la fase de Build y a la vez comunicarse con el contenedor "*firefox-container*" que acabamos lanzar. Para lanzar este contenedor vamos a crear otro Dockerfile, llamado "Dockerfile.e2e" que contendrá la siguiente declaración:

```
FROM node:carbon
WORKDIR ./
COPY ./package.json ./package-lock.json ./src ./
RUN npm install && npm install -g Mocha && npm install -g codeceptjs
CMD ["codeceptjs", "run", "--steps", "--verbose", "--config=./src/tests/e2eTests/config.js" ]
```

En este Dockerfile especificamos que la imagen parta de node:carbon. Copiaremos el contenido de la aplicación en el contenedor e instalaremos Mocha y codeceptjs, que son dependencias indispensables para la ejecución de los tests e2e que hemos programado. Por último, cuando el contenedor esté ejecutándose, lanzaremos un comando para comenzar los tests.

Para lanzar este contenedor, desde nuestra Pipeline, justo después de los mandatos para ejecutar el contenedor con el navegador, ejecutamos lo siguiente:

```
docker build -t portal-empleado-e2e-test -f Dockerfile.e2e .
docker run --net mynet -p 6666:6666 -v /$(pwd)/e2eTests:/e2eTests -v /$(pwd)/package.json:/package.json -v /$(pwd)/src:/src --rm portal-empleado-e2e-test
```

Ahora al lanzarse este contenedor llamado "portal-empleado-e2e-test" debería comunicarse con el contenedor "*firefox-container*" y con el contenedor "portal-empleado" y tratar de validar los tests que hemos desarrollado.

Veamos un caso de test E2E que hemos preparado para validar que un usuario puede iniciar sesión correctamente y acceder al panel principal de la aplicación:

```
Scenario('Navego a la pagina inicial', I => {
  I.amOnPage('/');
  I.see('Acceder');
  I.fillField("username",normalUserCredentials.username)
  I.fillField("password",normalUserCredentials.password)
  I.click('Acceder')
  I.see("MENÚ PRINCIPAL")
});
```

Como vemos, al igual que los tests de integración que hemos programado con Mocha, codeceptjs también nos permite desarrollar tests bastante descriptivos pues permiten saber qué estamos validando simplemente leyendo el código de manera natural.

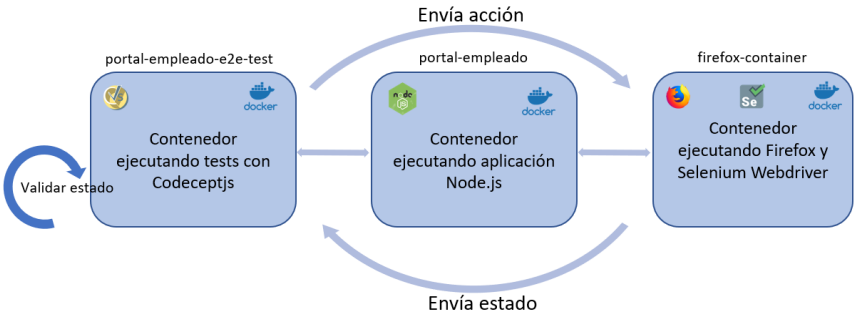


Figura 12. Flujo de comunicación entre los nodos encargados del *testing* E2E.

Cada test define una cabecera o escenario donde figura una breve descripción de lo que se desea validar. Seguidamente, se ejecutan las instrucciones del caso de test. Todas estas instrucciones que se están ejecutando en nuestro contenedor "portal-empleado-e2e-test" serán enviadas a la API que está a la escucha en el contenedor "*firefox-container*". Dicha API interpretará la petición, ejecutará una determinada acción y enviará el estado en el que se encuentra el documento actual del navegador de vuelta al contenedor que está ejecutando los tests para que pueda decidir si dicho test ha sido válido o no.

3.5. Despliegue Continuo

Llegamos a las fases Clean y Deploy. En la fase Clean lo que haremos es liberar recursos de todos los contenedores que hemos ido creando en las fases anteriores para ejecutar los tests. Primeramente liberaremos el contenedor utilizado para crear el navegador necesario para los tests E2E y por último cerraremos el contenedor que está corriendo nuestra aplicación sobre la que hemos ejecutado todos los tests.

```
docker stop firefox-container && docker rm firefox-container
docker stop portal-empleado && docker rm portal-empleado
```

Finalmente, tenemos la fase de despliegue (Deploy). Si la Pipeline ha llegado hasta esta fase significa que las fases anteriores se han ejecutado correctamente; en consecuencia, nuestra nueva versión está lista para desplegarse.

Como ya vimos en el capítulo anterior cuando estudiábamos las tecnologías a utilizar, vamos a tener dos versiones de la misma aplicación coexistiendo simultáneamente. Para ello vamos a utilizar un balanceador de carga que se situará por encima de las aplicaciones y será el encargado de discriminar a qué versión debe transferirse la petición.

3.5.1. Configuración de HAProxy en capa 7

Nuestro balanceador de carga será el encargado de recibir todas las peticiones de los usuarios de la aplicación. Estará actuando en el puerto 80 de nuestro host. Cuando reciba una petición, al estar configurado en capa 7, vamos a ser capaces de leer la cabecera de la petición que nos ha llegado.

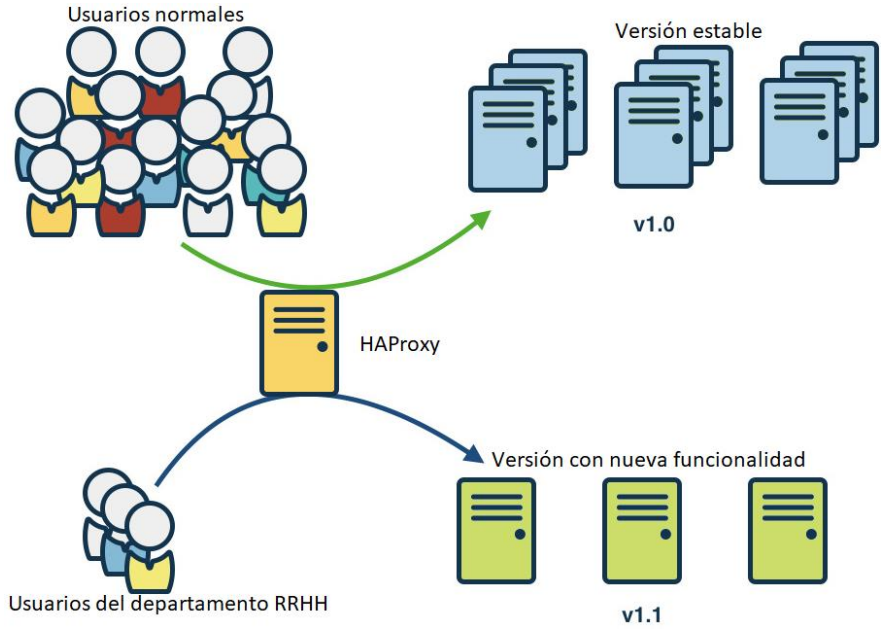


Figura 13. Diagrama de enrutamiento de peticiones discriminando por rol del usuario.

Una cabecera HTTP puede constar de la siguiente información:

```
    GET /dashboard/nominas/1 HTTP/1.1
    Host: net.tutsplus.com
    User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.1.5)
Gecko/20091102 Firefox/3.5.5 (.NET CLR 3.5.30729)
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
    Accept-Language: en-us,en;q=0.5
    Accept-Encoding: gzip,deflate
    Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
    Keep-Alive: 300
    Connection: keep-alive
    Cookie: is_rrhh=true;
```

```
Pragma: no-cache
Cache-Control: no-cache
```

Si nos fijamos, hay un dato denominado Cookie. En nuestro caso vamos a utilizar este campo de la cabecera HTTP como discriminador en HAProxy.

Contando con una instalación limpia de HAProxy, vamos a proceder a configurarlo para que pueda cumplir el objetivo que buscamos. HAProxy dispone de un fichero haproxy.cfg alojado en el directorio /etc/haproxy/ que contiene, en un lenguaje declarativo, la funcionalidad que debe desempeñar el balanceador una vez que su servicio se encuentre activo. La configuración que vamos a utilizar es la siguiente:

```
global
   maxconn 1000
defaults
   mode http
   timeout connect 30s
   timeout client 30s
   timeout server 30s
frontend http
   bind *:80
   acl cookie_rrhh hdr_sub(cookie) is_rrhh
   use_backend backend_rrhh if cookie_rrhh
   default_backend backend_normal

backend backend_rrhh
  server server2 10.0.0.4:4000

backend backend_normal
   server server1 10.0.0.4:3000
```

Fijándonos en las declaraciones de la configuración del balanceador, vemos que tiene varios apartados.

En primer lugar, en el apartado global definimos un límite máximo de conexiones simultáneas que aceptará el balanceador.

En el segundo apartado (defaults) establecemos varios parámetros referentes a los límites de tiempo de espera y establecemos el modo en el que va a ejecutarse HAProxy. Como en nuestro caso vamos a trabajar en capa 7, pondremos *mode http*. Si quisiéramos trabajar en capa 4 utilizaríamos *mode tcp*, pero en este último caso no podríamos leer las cabeceras HTTP de las peticiones.

En el tercer apartado se define la capa externa del balanceador. Aquí se especifica el puerto por el que van a entrar las peticiones (en nuestro caso, por el puerto 80). En este punto también especificaremos la regla para discriminar la petición y mandarla a una versión de aplicación u otra. En nuestra aplicación Node.js, hemos desarrollado un middleware que se ejecuta encadenado a cada petición. Este middleware se encarga de comprobar el rol del usuario que está

logeado y graba una cookie denominada *is_rrhh* si dicho usuario tiene rol de Recursos Humanos:

```
helpers.isLoggedIn = (req, res, next) => {
    res.clearCookie("is_rrhh")
    if(req.user!=null && req.user.IsRRHH){
        res.cookie("is_rrhh",true)
    }
    if (req.isAuthenticated() ) {
        return next()
    }
    else {
        return res.redirect('/auth/login')
    }
}
```

Como primer paso eliminamos la cookie de la cabecera, ya tenemos la cabecera limpia. En la primera sentencia condicional se puede ver como se comprueba si el usuario de la sesión tiene el rol de Recursos Humanos, en caso afirmativo grabamos la cookie.

Y desde HAProxy:

```
acl cookie_rrhh hdr_sub(cookie) is_rrhh
use_backend backend_rrhh if cookie_rrhh
default_backend backend_normal
```

Con la primera instrucción captamos la cookie "*is_rrhh*" de la cabecera de la petición. Con la segunda instrucción estamos diciendo que, si hemos conseguido obtener la cookie de la cabecera, significa que el usuario es personal de recursos humanos, en consecuencia redirigimos la petición al *backend* denominado *backend_rrhh*. Sin embargo, si no conseguimos capturar la cookie, significará que el usuario no es personal de recursos humanos, en consecuencia, redirigiremos a *backend_normal*.

Por último, configuramos los *backends* a los que va a redirigir las peticiones nuestro balanceador de carga:

```
backend backend_rrhh
  server server2 10.0.0.4:4000
backend backend_normal
   server server1 10.0.0.4:3000
```

Por una parte, tenemos el *backend* al que redirigiremos a los usuarios de Recursos Humanos. Especificaremos la IP privada de la máquina y el puerto (4000) donde estará publicada la versión *Green* de la aplicación. Por otra parte, tenemos el *backend* al que redirigiremos al resto de usuarios; la versión *Blue* será accesible por el puerto 3000.

4. CONCLUSIONES

Entre los objetivos que nos marcamos en un principio estaba el de realizar un análisis de metodologías y herramientas de Integración, Entrega y Despliegue Continuo con la finalidad de adquirir conocimiento, pero sobre todo criterio, para en un futuro tener la experiencia y poder decidir qué tecnologías aplicar en nuevos casos de uso a los que nos enfrentemos.

No solo hemos cumplido estas capacidades, sino que hemos conseguido aplicarlas a un caso de uso real con éxito. Paralelamente al desarrollo de este trabajo hemos ido implementando una aplicación web en Node.js que nos ha servido de ejemplo para probar todo tipo de herramientas que hemos estado analizando.

A lo largo de este trabajo hemos seleccionado tecnologías y estrategias para gestionar el control de versiones del código de una aplicación. Para validar la correcta funcionalidad y calidad del código hemos usado *testing* de integración y *End-to-End*. Utilizamos tecnologías de contenedores para crear entornos de ejecución y balanceadores de carga para administrar la redirección del tráfico. Finalmente integramos todo el sistema en un único servicio totalmente automatizado de modo que, solamente publicando un cambio en el repositorio de código, se dispara todo el flujo de eventos que hemos diseñado.

Todo este sistema y flujo de trabajo que hemos creado es utilizado en la metodología DevOps, la cual busca aumentar las frecuencias de despliegue de nuevas versiones, disminuir la tasa de errores en el código y minimizar el tiempo de recuperación en caso de que encontremos algún error en nuevas versiones.

A día de hoy, la aplicación que hemos ido desarrollando todavía no se ha completado. Sin embargo, gracias al sistema de trabajo implantado y al conjunto de prácticas empleadas, la aplicación ya se encuentra en producción con gran parte de su funcionalidad disponible para los usuarios, mientras gradualmente vamos integrando y liberando nuevas versiones.

5. REFERENCIAS

[1] Martin Fowler. Continuous Integration. https://martinfowler.com/articles/continuousIntegration.html 1 mayo, 2006.

[2] Leire Iturregi. Introducción a la automatización de tests E2E con Cypress.io. https://blog.irontec.com/introduccion-automatizacion-tests-e2e-cypress-io/ 15 enero, 2019.

[3] Ali Raza. Puppet vs. Chef vs. Ansible vs. Saltstack. https://www.intigua.com/blog/puppet-vs.-chef-vs.-ansible-vs.-saltstack 27 septiembre, 2016.

[4] Vagrant. Documentación. https://www.vagrantup.com/docs/

[5] RedHat. ¿Qué es Docker?. https://www.redhat.com/es/topics/containers/what-is-docker

[6] Jason Skowronski. Intro to deployment strategies. https://dev.to/mostlyjason/intro-to-deployment-strategies-*Blue-Green-Canary*-and-more-3a3 21 noviembre, 2018.

[7] W3Schools. Javascript versions. https://www.w3schools.com/js/js_versions.asp

[8] Ruby-lang. Ruby releases. https://www.ruby-lang.org/en/downloads/releases/

[9] By Ben. Javascript unit testing frameworks: Comparing Jasmine, Mocha, AVA, Tape and Jest. https://raygun.com/blog/javascript-unit-testing-frameworks/ 25 mayo, 2017.

[10] Adrian Lewis. Top 5 Most Rated Node.js Frameworks for End-to-End Web Testing. https://medium.com/@adrian_lewis/top-5-most-rated-node-js-frameworks-for-end-to-end-web-testing-f8ebca4e5d44 13 marzo, 2017.

[11] Wikipedia. PhantomJS. https://en.wikipedia.org/wiki/PhantomJS

[12] SeleliumHQ. Selenium Documentation. https://www.seleniumhq.org/docs/03_webdriver.jsp

[13] Federico Toledo. Travis-CI para integración continua. https://www.federico-toledo.com/travis-ci-para-integracion-continua/ 21 junio, 2018.

[14] CircleCI. CircleCI Documentation. https://circleci.com/docs/2.0/getting-started/

[15] Jenkins. Jenkins Documentation. https://jenkins.io/doc/

[16] Jenkins, Blue Ocean Documentation. https://jenkins.io/doc/book/blueocean/